

# SCR\*: A Toolset for Specifying and Analyzing Requirements

Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw  
Center for High Assurance Computer Systems  
Naval Research Laboratory  
Washington, DC 20375

## Abstract

*A set of CASE tools is described for developing formal requirements specifications expressed in the SCR (Software Cost Reduction) tabular notation. The tools include an editor for building the specifications, a consistency checker for testing the specifications for consistency with a formal requirements model, a simulator for symbolically executing the specifications, and a verifier for checking that the specifications satisfy selected application properties. As background, the SCR method for specifying requirements is reviewed, and a formal requirements model is introduced. Examples are presented to illustrate the tools.*

## 1 Introduction

*High assurance computer systems* are computer systems where compelling evidence is required that the system delivers its services in a manner that satisfies certain critical properties. Examples of high assurance systems include military command and control systems, nuclear power plants, telephone networks, medical systems (e.g., patient monitoring systems), air traffic control systems, and flight control systems. Critical properties that such systems must enforce include *security properties*, which prevent unauthorized disclosure, modification, and withholding of sensitive information (see, e.g., [11]); *safety properties*, which prevent unintended events that result in death, injury, illness, or damage to or loss of property; and *real-time properties*, which require the system to deliver its results within specified time intervals (see, e.g., [18]).

A promising approach for building high assurance systems is to apply formal methods. According to a recent study, formal methods are “beginning to be used seriously and successfully by industry...to develop systems of significant scale and importance” [6]. Especially important for developing high assurance systems are formal methods for specifying requirements. Studies have shown that a large portion of the most serious errors in safety-critical systems are requirements

errors (see, e.g., [19]). Formal specification can reduce requirements errors by reducing ambiguity and imprecision and by making instances of inconsistency and incompleteness obvious. Given a formal requirements specification, formal analysis can detect many classes of errors, some automatically.

One of the 12 methods included in the above study is the Software Cost Reduction (SCR) requirements method. Introduced originally to describe the functional requirements of software precisely and unambiguously [15, 16], the SCR method has been extended recently to describe system, rather than simply *software*, requirements and to represent both functional and nonfunctional (e.g., timing and accuracy) requirements [21, 24]. Designed for use by engineers, the SCR method has been applied successfully to a number of practical systems, including the A-7 aircraft’s Operational Flight Program [15, 1]; a submarine communications system [14]; and safety-critical components of two nuclear power plants, the Darlington plant in Canada [22] and a second plant in Belgium [5]. More recently, a version of the SCR method called CoRE [8] was used to document the requirements of Lockheed’s C-130J Operational Flight Program [9].

While the above applications of the SCR method rely on manual techniques, effective use of the method in industrial settings requires powerful, robust tool support. As observed in the formal methods study [6], tool support for formal methods, though currently weak and impoverished, must be “an integral part of a broader software development tool suite.” Further, one of the original developers of the SCR method and a leader in the certification of the Darlington software cites the need for tool support to make formal methods “practical” [22].

An important question is what form tool support should take. To answer this question, our group is developing a prototype toolset, called SCR\*, for constructing and analyzing formal requirements specifications. The toolset, which is coded in C++ and runs on SPARC workstations, includes a *specification editor* for building and displaying the requirements spec-

ifications, a *simulator* for symbolically executing the specifications, and *formal analysis* tools for testing the specifications for selected properties.

One analysis tool, called a *consistency checker* [10], tests a requirements specification for properties derived from our formal requirements model [12]. Because the requirements model describes the properties that all SCR-style requirements specifications must satisfy, the properties tested by the consistency checker are independent of a particular application. These properties are usually quite simple. They include proper syntax, type correctness, completeness (e.g., no missing cases), and deterministic behavior. A second analysis tool, called a *verifier*, checks the specification for critical application properties, such as safety properties, timing properties, and security properties. Because verification of application properties depends on a consistent (and complete) requirements specification, analysis using a verifier logically follows analysis with a consistency checker.

An industrial-strength formal method should have a formal (that is, mathematical) foundation and should be usable by engineers, scalable, and cost-effective. The tools described in this paper are an important component of such a method for requirements specification. They have a formal foundation, namely, our requirements model [12]. They are easy to use: after developing a requirements specification in the SCR notation, the engineer uses the tools to analyze the specification automatically and to execute it symbolically. They should scale up to handle practical applications: in two experiments, our tools detected several significant errors in a moderate-size requirements specification [13]. This evidence coupled with the high cost (several million dollars) of the Darlington certification effort, where error checking was done by hand, suggests that the tools are cost-effective.

The purpose of this paper is to introduce our toolset and the formal requirements model that provides its semantics. Section 2 reviews the SCR requirements method and introduces an example to illustrate the method. Section 3 summarizes our requirements model. Sections 4–7 describe the specification editor, consistency checker, simulator, and verifier that make up our toolset. Sections 8 and 9 describe related work and a process for developing requirements. Finally, Section 10 contains our conclusions.

## 2 Review of the SCR Method

**Background.** The purpose of a requirements document is to describe all acceptable system implementations [14]. To demonstrate a systematic method

for achieving this, the software requirements document for the A-7 aircraft’s Operational Flight Program was published in 1979. This document introduces many features associated with the SCR requirements method—the tabular notation, the underlying finite state machine model, and special constructs for specifying requirements, such as conditions and events, input and output data items, mode classes, and terms. Recently, a number of researchers, including Faulk [7, 8, 9], van Schouwen [24, 25], and Parnas [21], have extended and refined the original SCR method and strengthened the method’s formal foundation.

Faulk’s thesis in 1989 [7] provided formal definitions for parts of the A-7 model. In particular, it described the condition tables as total functions and the mode classes as finite state machines defined over events. A deficiency in the original A-7 document is that a mode class may be undefined in certain states; e.g., if no weapon was allocated, the Weapons mode class was undefined. Faulk’s model requires a mode class to be defined in every state; e.g., when no weapon is allocated, the Weapons mode class is in mode **None**. This not only makes analysis of the specifications more efficient; using mode classes to partition the state space also reduces the amount of detail, thus making the specifications easier to understand.

In 1990, van Schouwen’s thesis [24] presented a system-level requirements specification, also based on the A-7 model, for the Water Level Monitoring System (WLMS), part of the shutdown system for a nuclear power plant. The WLMS specification extends the SCR method from software requirements to system requirements and demonstrates the use of the method to describe a system’s functional requirements as well as its precision and timing requirements.

In 1991, Parnas and Madey introduced the Four Variable Model [21], which formalizes the innovations in van Schouwen’s thesis. The model, illustrated in Figure 1, describes the required system behavior in terms of quantities in the system’s environment.

**Four Variable Model.** The Four Variable Model describes the required system functions, timing, and accuracy as a set of mathematical relations on four sets of variables—monitored and controlled variables and input and output data items. A *monitored variable* represents an environmental quantity that influences system behavior, a *controlled variable* an environmental quantity that the system controls. A black box specification of required behavior is given as two relations, REQ and NAT, from the monitored to the controlled quantities. NAT, which defines the set of possible values, captures any natural constraints on the system behavior, such as those imposed by physi-

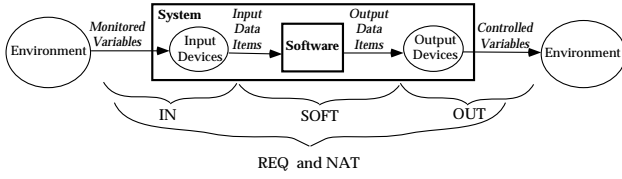


Figure 1: Four Variable Model.

cal laws and by the system environment. REQ defines additional constraints on the system to be built as relations the system must maintain between the monitored and the controlled quantities.

In the Four Variable Model, input and output data items, which represent the system’s input and output devices, are treated as resources. Input data items (e.g., sensors) are resources available to the system to sample the monitored quantities. The relation IN defines the mapping from the monitored quantities to the input data items. Similarly, the relation OUT defines the mapping from the output data items to the controlled quantities. The use of monitored and controlled quantities, rather than input and output data items, to define required behavior keeps the specification in the problem domain and allows a simpler specification.

Like the Four Variable Model, our requirements model can be used to describe both system requirements and software requirements. The former are described by defining REQ, the required relation between the monitored and controlled variables, the latter by describing SOFTREQ<sup>1</sup>, the required relation between the input and output data items. Thus, where appropriate, we use the term *input variable* to represent either a monitored variable or an input data item and the term *output variable* to represent either a controlled variable or an output data item.

The next section reviews the constructs and tabular notation used in SCR requirements specifications in terms of the Four Variable Model. Because our initial requirements model emphasizes the system’s functions, the discussion focuses on aspects of the Four Variable Model that describe functional behavior.

**SCR Constructs.** To specify the relations of the Four Variable Model in a practical and efficient manner, four other constructs, all introduced in the A-7 requirements document [15], are useful. These are modes, terms, conditions, and events. A *mode class* is a state machine, whose states are called *system modes* (or simply *modes*) and whose transitions are triggered by events. Complex systems are defined by more than

<sup>1</sup>In the Four Variable Model, SOFT represents the software implementation; by SOFTREQ, we mean the software requirements specification.

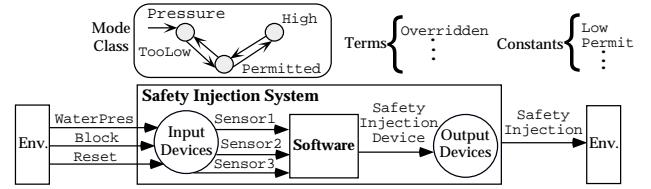


Figure 2: Requirements Spec. for Safety Injection.

one mode class, operating in parallel. A *term* is any function of input variables, modes, or other terms that helps make the specification concise. A *condition* is a predicate defined on one or more system entities (a system *entity* is an input or output variable, mode, or term) at some point in time. An *event* occurs when any system *entity* changes value. A special event, called an *input event*, occurs when an input variable changes value. Another special event, called a *conditioned event*, occurs if an event occurs when a specified condition is true.

To illustrate the SCR constructs, we consider a simplified version of the control system for safety injection described in [5]. The system uses three sensors to monitor water pressure and adds coolant to the reactor core when the pressure falls below some threshold. The system operator blocks safety injection by turning on a “Block” switch and resets the system after blockage by turning on a “Reset” switch. Figure 2 shows how SCR constructs could be used to specify the requirements of the control system. Water pressure and the “Block” and “Reset” switches are represented as monitored variables, **WaterPres**, **Block**, and **Reset**; safety injection as a controlled variable, **Safety Injection**; each sensor as an input data item; and the hardware interface between the control system software and the safety injection system as an output data item.

The specification illustrated in Figure 2 includes a mode class **Pressure**, a term **Overridden**, and several conditions and events. The mode class **Pressure**, an abstract model of the monitored variable **WaterPres**, contains three modes, **TooLow**, **Permitted**, and **High**. At any given time, the system must be in one of these modes. A drop in water pressure below a constant **Low** causes the system to enter mode **TooLow**; an increase in pressure above a larger constant **Permit** causes the system to enter mode **High**. The term **Overridden** is *true* if safety injection is blocked, *false* otherwise. An example of a condition in the specification is “**WaterPres** < **Low**”. Events are denoted by the notation “@T”. Two examples of events are the input event @T(**Block**=On) (the operator turns **Block** from **Off** to **On**) and the conditioned event @T(**Block**=On)

WHEN **WaterPres** < **Low** (the operator turns **Block** to **On** when water pressure is below **Low**).

**SCR Tables.** The A-7 requirements document [15] introduced a special tabular notation for writing specifications. The tabular notation facilitates industrial application of the SCR method: Not only do engineers find tables easy to understand and to develop; in addition, tables can describe large quantities of requirements data concisely. Among the tables in SCR specifications are condition tables, event tables, and mode transition tables. Each table defines a function.<sup>2</sup> A condition table describes an output variable or term as a function of a mode and a condition; an event table describes either as a function of a mode and an event. A mode transition table describes a mode as a function of another mode and an event.

While condition tables define total functions, event tables and mode transition tables may define partial functions, because some events cannot occur when certain conditions are true. For example, in the above system, the event @T(**Pressure**=**High**) WHEN **Pressure**=**TooLow** cannot occur, because starting from **TooLow**, the system can only enter **Permitted** when a state transition occurs.

Tables 1–3, each constructed with our toolset, are part of REQ, the system requirements specification for the above control system. The tables use the SCR bracketing notation to indicate the “class” of an object (e.g., **\*\*...\*\*** for a mode class, **\*...\*** for a mode, **%...%** for a monitored variable, **%%...%%** for a controlled variable, **!...!** for a term, and **\$...\$** for a non-numeric value). Future versions of the toolset will allow the user to change or omit this notation.

Table 1 is a mode transition table describing the mode class **Pressure** as a function of the current mode and the monitored variable **WaterPres**. Table 2 is an event table describing the term **Overridden** as a function of **Pressure**, **Block**, and **Reset**. Table 3 is a condition table describing the controlled variable **Safety Injection** as a function of **Pressure** and **Overridden**. Table 3 states, “If **Pressure** is **High** or **Permitted** or if **Pressure** is **TooLow** and **Overridden** is *true*, then **Safety Injection** is **Off**; if **Pressure** is **TooLow** and **Overridden** is *false*, then **Safety Injection** is **On**.” The notation “@T(Inmode)” in a row of an event table describes system entry into the mode in that row; for example, “@T(Inmode)” in the first row of Table 2 means, “If the system enters **High**, then **Overridden** becomes *false*.”

<sup>2</sup>Although SCR specifications can be nondeterministic, our initial model is restricted to deterministic systems.

The screenshot shows a software window titled "SafetyInjection.Spec : \*\*Pressure\*\*". It has a menu bar with "Table", "Edit", "View", "Tools", and "Help". Below the menu bar, there are fields for "Name: \*\*Pressure\*\*", "Table Type: Mode Transition", and "Class: Mode Class". The main area is divided into three columns: "Source Mode", "Events", and "Destination Mode".

Source Mode	Events	Destination Mode
*TooLow*	@T(%WaterPres% >= \$Low\$)	*Permitted*
*Permitted*	@T(%WaterPres% < \$Low\$)	*TooLow*
*Permitted*	@T(%WaterPres% >= \$Permit\$)	*High*
*High*	@T(%WaterPres% < \$Permit\$)	*Permitted*

Table 1: Mode Transition Table for **Pressure**.

The screenshot shows a software window titled "SafetyInjection.Spec : !Overridden!". It has a menu bar with "Table", "Edit", "View", "Tools", and "Help". Below the menu bar, there are fields for "Name: !Overridden!", "Table Type: Event", and "Class: Term". The "Mode Class" field is set to "\*\*Pressure\*\*". The main area is divided into two columns: "Modes" and "Events".

Modes	Events
*High*	Never @T(Inmode)
*TooLow*, *Permitted*	@T(%Block%=\$ON\$) WHEN %Reset%=\$OFF\$ @T(Inmode) OR @T(%Reset%=\$ON\$)
!Overridden!=	\$TRUE\$ \$FALSE\$

Table 2: Event Table for **Overridden**.

The screenshot shows a software window titled "SafetyInjection.Spec : %%Safety\_Injection%%". It has a menu bar with "Table", "Edit", "View", "Tools", and "Help". Below the menu bar, there are fields for "Name: %%Safety\_Injection%%", "Table Type: Condition", and "Class: Controlled Variable". The "Mode Class" field is set to "\*\*Pressure\*\*". The main area is divided into two columns: "Modes" and "Conditions".

Modes	Conditions
*High*, *Permitted*	\$TRUE\$ \$FALSE\$
*TooLow*	!Overridden! Not !Overridden!
%Safety_injection%%	\$OFF\$ \$ON\$

Table 3: Condition Table for **Safety Injection**.

### 3 Formal Requirements Model

Although earlier requirements models, namely, Faulk’s automaton model [7], the model underlying van Schouwen’s specification [24, 25], and the Four-Variable Model, define some aspects of the SCR requirements method, these models are too abstract to provide a formal basis for our tools. To provide a precise and detailed semantics for the SCR method, our model represents the system to be built as a finite state automaton and describes the input and output variables, conditions, events, and other constructs that make up an SCR specification in terms of that automaton. Our automaton model is a special case of the Four Variable Model. One significant difference is that the Four Variable Model represents naturally continuous environmental quantities, such as pressure and temperature, as continuous variables, whereas our model represents these quantities as discrete variables.

Our requirements model defines sets of modes, entity names, values, and data types and a special function  $TY$ , which maps an entity to its legal values. The model defines system state in terms of the entities, a condition as a predicate on the system state, and an input event as a change in an input variable that triggers a new system state. It then shows how a set of functions, called table functions, can be derived from the SCR tables. These table functions define the transform  $T$ , a special case of REQ (and SOFTREQ), which maps the current state and an input event to a new state. We present below excerpts from our requirements model [12] along with examples taken from the system requirements specification for the simple control system introduced above.

**System State.** We assume the existence of the following sets.

- $MS$  is the union of  $N$  pairwise disjoint sets, called *mode classes*. Each member of a mode class is called a *mode*.
- $TS$  is a union of data types, where each type is a nonempty set of values.
- $VS = MS \cup TS$  is the set of entity values.
- $RF$  is a set of entity names  $r$ .  $RF$  is partitioned into four subsets:  $MR$ , the set of mode class names;  $IR$ , the set of input variable names;  $GR$ , the set of term names; and  $OR$ , the set of output variable names. For all  $r \in RF$ ,  $TY(r) \subseteq VS$  is the type of the entity named  $r$ .

A *system state*  $s$  is a function that maps each entity name  $r$  in  $RF$  to a value. More precisely, for all  $r \in RF$ :

$s(r) = v$ , where  $v \in TY(r)$ . Thus, by assumption, in any state  $s$ , the system is in exactly one mode from each mode class, and each entity has a unique value.

*Example.* In the sample system, the set of entity names  $RF$  is defined by

$$RF = \{\text{Block}, \text{Reset}, \text{WaterPres}, \text{Pressure}, \text{SafetyInjection}, \text{Overridden}\}.$$

The type definitions include

$$\begin{aligned} TY(\text{Pressure}) &= \{\text{TooLow}, \text{Permitted}, \text{High}\} \\ TY(\text{WaterPres}) &= \{0, 1, 2, \dots, 2000\} \\ TY(\text{Overridden}) &= \{\text{true}, \text{false}\} \\ TY(\text{Block}) &= \{\text{On}, \text{Off}\}. \end{aligned}$$

**Conditions.** Conditions are defined on the values of entities in  $RF$ . A *simple condition* is *true*, *false*, or a logical statement  $r \odot v$ , where  $r \in RF$  is an entity name,  $\odot \in \{=, \neq, >, <, \geq, \leq\}$  is a relational operator, and  $v \in TY(r)$  is a constant value.<sup>3</sup> A *condition* is a logical statement composed of simple conditions connected in the standard way by the logical connectives  $\wedge$ ,  $\vee$ , and NOT.

**System (Software System).** A *system (software system)*  $\Sigma$  is a 4-tuple,  $\Sigma = (E^m, S, s_0, T)$ , where

- $E^m$  is a set of input events. A *primitive event* is denoted as  $@T(r = v)$ , where  $r$  is an entity in  $RF$  and  $v \in TY(r)$ . An *input event* is a primitive event  $@T(r = v)$ , where  $r \in IR$  is an input variable.
- $S$  is the set of possible system states.
- $s_0$  is a special state called the initial state.
- $T$  is the system transform, i.e., a function from  $E^m \times S$  into  $S$ .

**Events.** In addition to denoting primitive events, the “@T” notation also denotes basic and conditioned events. A *basic event* is denoted as  $@T(c)$ , where  $c$  is any simple condition. A *simple conditioned event* is denoted as  $@T(c) \text{ WHEN } d$ , where  $@T(c)$  is a basic event and  $d$  is a simple condition or a conjunction of simple conditions. Any basic event  $@T(c)$  can be expressed as the simple conditioned event  $@T(c) \text{ WHEN true}$ . A *conditioned event*  $e$  is composed of simple conditioned events connected by the logical connectors  $\wedge$  and  $\vee$ .

<sup>3</sup>Here as well as elsewhere in the formal model,  $v$  is defined as a constant to keep the notation simple. Our formal model eventually generalizes this definition:  $v$  may be a function defined on entities.

The logical statement represented by a simple conditioned event is defined by

$$@T(c) \text{ WHEN } d = \text{NOT } c \wedge c' \wedge d, \quad (1)$$

where the unprimed version of condition  $c$  denotes  $c$  in the old state and the primed version denotes  $c$  in the new state. Given  $c = r \odot v$ , we define  $c'$  as  $c' = (r \odot v)' = r' \odot v$ . Based on these definitions and the standard predicate calculus, any conditioned event can be expressed as a logical statement.

*Example.* Applying the definition in (1), the conditioned event  $@T(\text{Block}=\text{On}) \text{ WHEN } \text{Reset}=\text{Off}$  can be rewritten as  $\text{Block}' = \text{On} \wedge \text{Block} \neq \text{On} \wedge \text{Reset} = \text{Off}$ . This event occurs if both **Block** and **Reset** are **Off** in the old state and **Block** is **On** in the new state.

**Ordering the Entities.** To compute the value of an entity in the new state, the transform function may use the values of entities in both the old state and the new state. To describe the entities needed in the new state, each entity  $r$  is associated with a subset of  $RF$  called the *new state dependencies* set. Given entities  $r$  and  $\hat{r}$  in  $RF$ , we say that  $r$  *depends directly on*  $\hat{r}$  if  $\hat{r}$  is in  $r$ 's new state dependencies set. The “depends directly on” relation imposes a partial ordering on the set  $RF$ . Thus, the entities in  $RF$  can be ordered as a sequence  $R$ , where for all  $i$  and  $j$  such that  $r_i$  and  $r_j$  belong to  $R$ ,  $r_i$  depends directly on  $r_j$  implies that  $r_i$  follows  $r_j$  in  $R$  (that is,  $i > j$ ). The new state dependencies set for entity  $r_i$  in  $R$  is denoted as  $D_i^n$ .

*Example.* The condition table in Table 3 shows that the controlled variable **SafetyInjection** depends on two entities in the new state, the mode class **Pressure** and the term **Overridden**. Hence, the new state dependencies set for **SafetyInjection** contains **Pressure** and **Overridden**. The partial ordering of the entities based on dependencies in the new state is determined as follows: The three monitored variables are first because they only depend on changes in the environment. Next is the mode class **Pressure**, which depends on **WaterPres**. Next is the term **Overridden**, which depends on **Pressure** and two monitored variables, **Block** and **Reset**. The last entity in the partial ordering is **SafetyInjection**. A sequence  $R$  satisfying this partial ordering is

$$R = \langle \text{WaterPres}, \text{Block}, \text{Reset}, \text{Pressure}, \text{Overridden}, \text{SafetyInjection} \rangle.$$

In sequence  $R$ , **SafetyInjection** =  $r_6$ . Its new state dependencies set is  $D_6^n = \{\text{Pressure}, \text{Overridden}\}$ .

**Table Functions.** Each SCR table describes a *table function*, called  $F_i$ , which defines an output variable,

a term, or a mode class  $r_i$ . Each entity  $r_i$  defined by a table is associated with exactly one mode class,  $M_j$ ,  $1 \leq j \leq N$ . To represent the relation between an entity and a mode class, we define a function  $\mu$ , where  $\mu(i) = j$  iff entity  $r_i$  is associated with mode class  $M_j$ . Using this notation,  $M_{\mu(i)}$  denotes the mode class associated with entity  $r_i$ .

*Example.* The only mode class is **Pressure**. Hence,  $N = 1$  and  $M_1 = \text{Pressure}$ . Because all three entities defined by tables, namely,  $r_4 = \text{Pressure}$ ,  $r_5 = \text{Overridden}$ , and  $r_6 = \text{SafetyInjection}$ , are functions of **Pressure**, we have  $\mu(4) = \mu(5) = \mu(6) = 1$ .

Presented below for condition, event, and mode transition tables is a typical format and a description of how the table function is derived from a given table.

Modes	Conditions			
$m_1$	$c_{1,1}$	$c_{1,2}$	$\dots$	$c_{1,p}$
$m_2$	$c_{2,1}$	$c_{2,2}$	$\dots$	$c_{2,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$m_n$	$c_{n,1}$	$c_{n,2}$	$\dots$	$c_{n,p}$
$r_i$	$v_1$	$v_2$	$\dots$	$v_p$

Table 4: Condition Table—Typical Format.

**Condition Tables.** Table 4 shows a typical format for a condition table with  $n+1$  rows and  $p+1$  columns. Each condition table describes an output variable or term  $r_i$  as a relation  $\rho_i$  defined on modes, conditions, and values. More precisely,  $\rho_i = \{(m_j, c_{j,k}, v_k) \in M_{\mu(i)} \times C_i \times TY(r_i)\}$ , where  $C_i$  is a set of conditions defined on entities in  $RF$ .  $\rho_i$  has the following four properties:

1. The  $m_j$  and the  $v_k$  are unique.
2.  $\cup_{j=1}^n m_j = M_{\mu(i)}$  (All modes are included).
3. For all  $j$ :  $\vee_{k=1}^p c_{j,k} = \text{true}$  (Coverage: The disjunction of the conditions in each row of the table is *true*).
4. For all  $j, k, l$ ,  $k \neq l$ :  $c_{j,k} \wedge c_{j,l} = \text{false}$  (Disjointness: The conjunction of any pair of conditions in a row of the table is *false*).

These properties guarantee that  $\rho_i$  is a function.

To make explicit entity  $r_i$ 's dependencies on other entities, we consider an alternate form  $F_i$  of the function  $\rho_i$ . To define  $F_i$ , we require the new state dependencies set,  $D_i^n = \{y_{i,1}, y_{i,2}, \dots, y_{i,n_i}\}$ , where  $y_{i,1}$  is the entity name for the associated mode class. Based on  $D_i^n$  and  $\rho_i$ , we define  $F_i$  as

$$F_i(y_{i,1}, \dots, y_{i,n_i}) = \begin{cases} v_1 & \text{if } \vee_{j=1}^n (y_{i,1} = m_j \wedge c_{j,1}) \\ v_2 & \text{if } \vee_{j=1}^n (y_{i,1} = m_j \wedge c_{j,2}) \\ \vdots & \\ v_p & \text{if } \vee_{j=1}^n (y_{i,1} = m_j \wedge c_{j,p}). \end{cases}$$

The function  $F_i$  is called a *condition table* function. The four properties guarantee that  $F_i$  is total.

*Example.* Based on the new state dependencies set  $D_6^n$  and Table 3, the condition table function for **Safety Injection**, denoted  $F_6$ , is defined by

$$\text{SafetyInjection}' = F_6(\text{Pressure}, \text{Overridden}) = \begin{cases} \text{Off} & \text{if } \text{Pressure} = \text{High} \vee \text{Pressure} = \text{Permitted} \vee (\text{Pressure} = \text{TooLow} \wedge \text{Overridden} = \text{true}) \\ \text{On} & \text{if } \text{Pressure} = \text{TooLow} \wedge \text{Overridden} = \text{false} \end{cases}$$

Modes	Events			
$m_1$	$e_{1,1}$	$e_{1,2}$	$\dots$	$e_{1,p}$
$m_2$	$e_{2,1}$	$e_{2,2}$	$\dots$	$e_{2,p}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$m_n$	$e_{n,1}$	$e_{n,2}$	$\dots$	$e_{n,p}$
$r_i$	$v_1$	$v_2$	$\dots$	$v_p$

Table 5: Event Table—Typical Format.

**Event Tables.** Table 5 illustrates a typical format for an event table with  $n+1$  rows and  $p+1$  columns. Each event table describes an output variable or term  $r_i$  as a relation  $\rho_i$  between modes, conditioned events, and values, i.e.,  $\rho_i = \{(m_j, e_{j,k}, v_k) \in M_{\mu(i)} \times E_i \times TY(r_i)\}$ , where  $E_i$  is a set of conditioned events defined on entities in RF.  $\rho_i$  has the following two properties:

1. The  $m_j$  and the  $v_k$  are unique.
2. For all  $j, k, l$ ,  $k \neq l$ :  $e_{j,k} \wedge e_{j,l} = \text{false}$  (Disjointness: The conjunction of each pair of events in a row of the table is *false*).

These properties and assumptions on input events guarantee that  $\rho_i$  is a function. As with condition tables, we make explicit  $r_i$ 's dependency on other entities by defining an alternate form  $F_i$  of the function  $\rho_i$ . To define  $F_i$ , we require both the new state dependencies set  $D_i^n$  and an *old state dependencies* set  $D_i^o = \{x_{i,1}, x_{i,2}, \dots, x_{i,m_i}\}$ , where  $D_i^o \subseteq RF$  contains the entities needed in the old state to compute  $r_i$  and  $x_{i,1}$  is the entity name for the associated mode class. Based on  $D_i^o$ ,  $D_i^n$ , and  $\rho_i$ ,  $F_i$  is defined by

$$F_i(x_{i,1}, \dots, x_{i,m_i}, y'_{i,1}, \dots, y'_{i,n_i}) = \begin{cases} v_1 & \text{if } \bigvee_{j=1}^n (x_{i,1} = m_j \wedge e_{j,1}) \\ v_2 & \text{if } \bigvee_{j=1}^n (x_{i,1} = m_j \wedge e_{j,2}) \\ \vdots & \\ v_p & \text{if } \bigvee_{j=1}^n (x_{i,1} = m_j \wedge e_{j,p}). \end{cases}$$

The function  $F_i$  is called an *event table* function.

*Example.* Using Table 2, the old and new dependencies sets for the event table **Overridden** can be derived. These are  $D_5^o = D_5^n =$

$\{\text{Block}, \text{Reset}, \text{Pressure}\}$ . Given  $D_5^o$ ,  $D_5^n$ , and Table 2, the event table function for **Overridden** is defined by

$$\text{Overridden}' = F_5(\text{Block}, \text{Reset}, \text{Pressure}, \text{Block}', \text{Reset}', \text{Pressure}') = \begin{cases} \text{true} & \text{if } (\text{Pressure} = \text{TooLow} \wedge \text{Block}' = \text{On} \wedge \text{Block} = \text{Off} \wedge \text{Reset} = \text{Off}) \vee (\text{Pressure} = \text{Perm.} \wedge \text{Block}' = \text{On} \wedge \text{Block} = \text{Off} \wedge \text{Reset} = \text{Off}) \\ \text{false} & \text{if } (\text{Pressure} = \text{TooLow} \wedge \text{Reset}' = \text{On} \wedge \text{Reset} = \text{Off}) \vee (\text{Pressure} = \text{Perm.} \wedge \text{Reset}' = \text{On} \wedge \text{Reset} = \text{Off}) \vee (\text{Pressure}' = \text{High} \wedge \text{Pressure} \neq \text{High}) \vee (\text{Pressure}' = \text{TooLow} \wedge \text{Pressure} \neq \text{TooLow}) \vee (\text{Pressure}' = \text{Perm.} \wedge \text{Pressure} \neq \text{Perm.}) \end{cases}$$

Old Mode	Event	New Mode
$m_1$	$e_{1,1}$	$m_{1,1}$
	$e_{1,2}$	$m_{1,2}$
	$\dots$	$\dots$
	$e_{1,k_1}$	$m_{1,k_1}$
$\dots$	$\dots$	$\dots$
$m_n$	$e_{n,1}$	$m_{n,1}$
	$e_{n,2}$	$m_{n,2}$
	$\dots$	$\dots$
	$e_{n,k_n}$	$m_{n,k_n}$

Table 6: Mode Transition Table—Typical Format.

**Mode Transition Tables.** Table 6 shows a typical format for a mode transition table for an entity  $r_i$  that names a mode class  $M_{\mu(i)}$ . The table describes  $r_i$  as a relation  $\rho_i = \{(m_j, e_{j,k}, m_{j,k}) \in M_{\mu(i)} \times E_i \times M_{\mu(i)}\}$ , where  $E_i$  is a set of conditioned events defined on entities in RF.  $\rho_i$  has the following four properties:

1. The  $m_j$  are unique.
2. For all  $k \neq k'$ ,  $m_{j,k} \neq m_{j,k'}$ , and for all  $j$  and for all  $k$ ,  $m_j \neq m_{j,k}$ .
3. For all  $j, k, k'$ ,  $k \neq k'$ :  $e_{j,k} \wedge e_{j,k'} = \text{false}$  (Disjointness: The conjunction of any pair of conditioned events in a row of the table is *false*).
4. For all  $m \in M_{\mu(i)}$ , there exists  $j$  such that  $m_j = m$  or there exist  $j$  and  $k$  such that  $m_{j,k} = m$  (Each mode in the mode class is in either  $\rho_i$ 's domain or its image).

These properties and assumptions on input events guarantee that  $F_i$  is a function. It is easy to show that a mode transition table with the format shown in Table 6 can be expressed in the format shown for an event table. Hence, a mode transition table can be expressed as an event table function  $F_i$ .

*Example.* Based on Table 1, the old and new dependencies sets for the mode class **Pressure** are defined by  $D_i^o = \{\text{WaterPres}, \text{Pressure}\}$  and  $D_i^n =$

$\{\text{WaterPres}\}$ . Given  $D_i^o$ ,  $D_i^n$ , and Table 1, the table function for **Pressure** is defined by

$$\text{Pressure}' = F_4(\text{Pressure}, \text{WaterPres}, \text{WaterPres}') =$$

$$\left\{ \begin{array}{ll} \text{TooLow} & \text{if } \text{Pressure} = \text{Perm.} \wedge \text{WaterPres}' < \text{Low} \wedge \text{WaterPres} \not< \text{Low} \\ \text{High} & \text{if } \text{Pressure} = \text{Perm.} \wedge \text{WaterPres}' \geq \text{Permit} \wedge \text{WaterPres} \not\geq \text{Permit} \\ \text{Perm.} & \text{if } (\text{Pressure} = \text{TooLow} \wedge \text{WaterPres}' \geq \text{Low} \wedge \text{WaterPres} \not\geq \text{Low}) \vee \\ & (\text{Pressure} = \text{High} \wedge \text{WaterPres}' < \text{Permit} \wedge \text{WaterPres} \not< \text{Permit}). \end{array} \right.$$

## 4 Specification Editor

To create, modify, or display a specification, the user invokes the specification editor. As illustrated in Figure 3, the editor lists the tables and dictionaries that make up a specification in a window labeled “Specification Contents”. The tables are organized into event, mode transition, and condition tables. By double clicking on any table class listed in the Specification Contents, e.g., “Event Table”, the user can list all tables of that class in the specification. In the example shown in Figure 3, only a single event table exists – the one defining the term !Overridden!. Double clicking on “!Overridden!” displays Table 2.

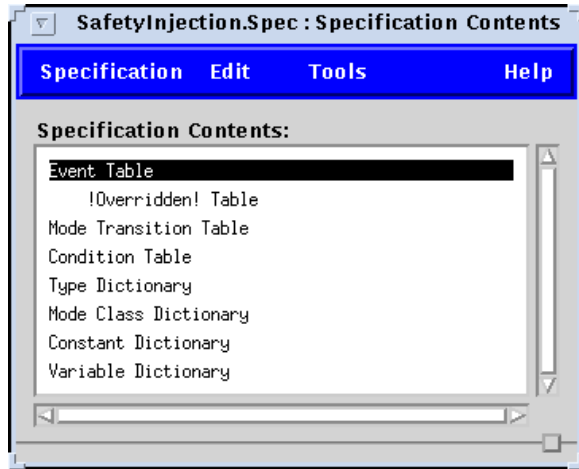


Figure 3: Contents of a Specification.

Each specification contains four dictionaries: the constant, type, mode class, and variable dictionaries. The *constant dictionary* assigns value to constants in the specification, while the *type dictionary* describes each user-defined type in terms of some base type. For each mode class in the specification, the *mode class dictionary* lists the modes in the class along with its

initial value. For each term and monitored and controlled variable in the specification, the *variable dictionary* lists the variable’s type, initial value, and accuracy requirements.



Figure 4: Constant Dictionary.



Figure 5: Type Dictionary.

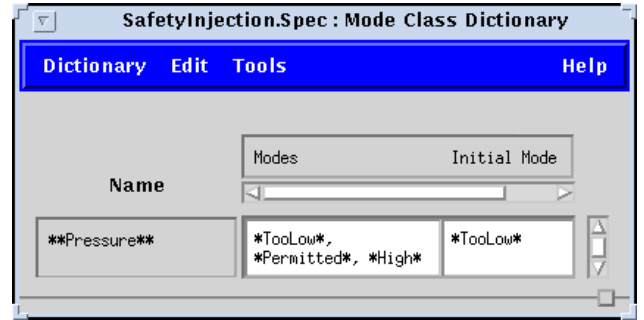


Figure 6: Mode Class Dictionary.

Figures 4–7 show the constant, type, mode class, and variable dictionaries for the simple safety injection system introduced above. The constant dictionary in Figure 4 defines two constants, \$Low\$=900 and \$Permit\$=1000. The type dictionary in Figure 5 describes two user-defined types: +Pressure+, a non-negative integer not exceeding 2000, and +Switch+, an enumerated type with values \$OFF\$ and \$ON\$. The mode class dictionary in Figure 6 defines the component modes and initial value of \*\*Pressure\*\*, the single mode class in the specification. Finally, the variable dictionary in Figure 7 defines the type, initial value, and accuracy requirements of the five variables in the specification.

Name	Class	Type	Initial Value	Accuracy
%Block%	Monitored Variable	+Switch+	\$OFF\$	N/A
%Reset%	Monitored Variable	+Switch+	\$OFF\$	N/A
%WaterPres%	Monitored Variable	+Pressure+	14	0.05%
!Overridden!	Term	+Boolean+	\$FALSE\$	N/A
%%Safety_Injection%%	Controlled Variable	+Switch+	\$OFF\$	N/A

Figure 7: Variable Dictionary.

## 5 Consistency Checker

Listed below are consistency checks derived from our formal requirements model. These checks, which determine whether the specifications are well-formed, are independent of a particular system state. They are a form of static analysis: they can be performed directly on the information in the tables and the dictionaries. For example, the Disjointness and Coverage checks on a condition table can be completed using only the information in the table and the relevant type definitions in the type dictionary.

- **Proper Syntax.** Each component of the specification has proper syntax. For example, each condition and event is well-formed.
- **Type Correctness.** Each variable has a defined type, and all type definitions are satisfied.
- **Completeness of Variable and Mode Class Definitions.** The value of each controlled variable, term, and mode class is defined. (Most variables will be defined by tables, but standard mathematical definitions may be given for some controlled variables and terms.)
- **Reachability.** Every mode in a mode class is reachable. (This property can be easily checked by analyzing the mode transition tables.)
- **Initial Values.** Initial values are provided for all mode classes, monitored variables, and all terms and controlled variables not defined by condition tables. (Initial values are not needed for variables defined by condition tables, since they can be derived from the tables. If initial values are pro-

vided, they must be consistent with the condition table definitions.)

- **Disjointness.** To make the specifications deterministic, each condition, event, and mode transition table must satisfy the Disjointness property. That is, in a given state, each controlled variable and term has a unique value, and if a state transition occurs, the new state is unique.
- **Coverage.** Each condition table satisfies the Coverage property. That is, each variable described by a condition table is defined everywhere in its domain.
- **Lack of Circularity.** No circular dependencies exist.

Clearly, the user must invoke some checks before others. For example, checks for proper syntax must precede type checking, and type checking should precede checks that the Coverage property is satisfied.

A prototype consistency checker that performs most of the above checks has been implemented. The user may apply the consistency checker either to an individual table or to selected components of the complete specification. In the first two examples included in this section, consistency checking was applied to an individual table (see Figures 8 and 9). In the third, “All Checks” were applied to the complete specification (see Figure 10).

**Examples.** Figure 8 shows the results of applying consistency checking to an erroneous version of the mode transition table in Table 1. The consistency checker finds two type errors: In the first column of the table’s first and second rows, the mode names are

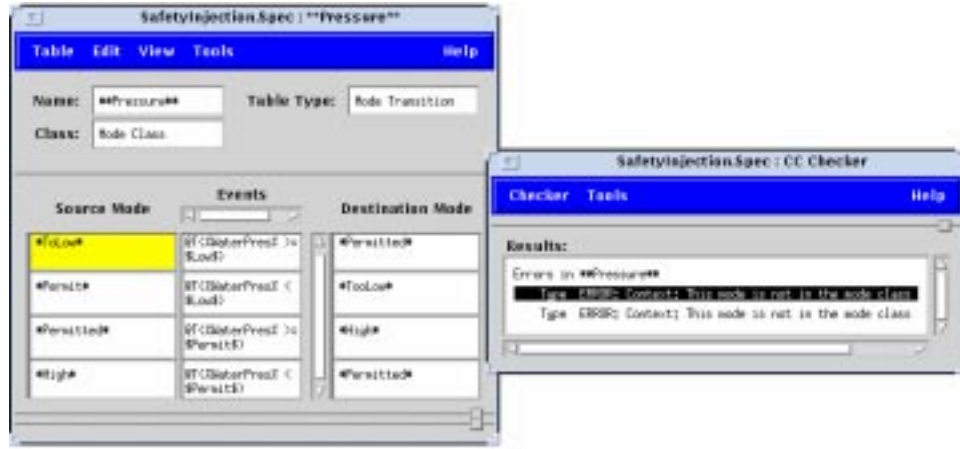


Figure 8: Type Errors in a Mode Transition Table.



Figure 9: Coverage and Disjointness Errors in a Condition Table.

misspelled (i.e., “ToLow” instead of “TooLow” and “Permit” instead of “Permitted”). By double clicking on each error description in the Results box of the CC Checker, the user can localize the error. For example, in Figure 8, the user double clicked on the first error listed; to indicate the corresponding type error, the simulator highlights “ToLow” in row 1 of the table defining **\*\*Pressure\*\***.

Shown in Figure 9 is a variation of Table 3, which omits the “NOT” in the rightmost column of the second row. Checking this condition table for consistency reveals two errors: The table’s second row violates both Coverage (**Overridden**  $\vee$  **Overridden**  $\neq$  *true*) and Disjointness (**Overridden**  $\wedge$  **Overridden**  $\neq$  *false*). Double clicking on an error description in the Results box of the Consistency Checker will display the table in which the error occurs with the error’s location highlighted (see row 2 of the condition table in Figure 9). In the case of Disjointness, the tool highlights the two overlapping table entries. In the case of Coverage, the tool highlights the row that is in error.

Disjointness, the second property required of event tables, is violated if two events in a row, say  $e$  and  $e'$ , overlap, i.e.,  $e \wedge e' \neq \text{false}$ . Figure 10 contains a variation of the event table in Table 2. Running the consistency checker detects a Disjointness error.

In checking for Disjointness, the consistency checker evaluates the expression,  $[\text{@T}(\text{Block} = \text{On}) \text{ WHEN } \text{Reset} = \text{Off}] \wedge [\text{@T}(\text{Block} = \text{On}) \vee \text{@T}(\text{Reset} = \text{On})]$ . Rewriting this expression as a disjunction produces  $[\text{@T}(\text{Block}=\text{On}) \text{ WHEN } \text{Reset}=\text{Off} \wedge \text{@T}(\text{Block}=\text{On})] \vee [\text{@T}(\text{Block}=\text{On}) \text{ WHEN } \text{Reset}=\text{Off} \wedge \text{@T}(\text{Reset}=\text{On})]$ . Applying [1] to the first clause of the disjunction, we have  $[\text{Block}' = \text{On} \wedge \text{Block}=\text{Off} \wedge \text{Reset}=\text{Off}] \wedge [\text{Block}' = \text{On} \wedge \text{Block}=\text{Off}]$ . This simplifies to  $\text{Block}' = \text{On} \wedge \text{Block}=\text{Off} \wedge \text{Reset}=\text{Off}$ . Because this expression does not equal *false*, the specified behavior is nondeterministic. This means that, if in **TooLow** or **Permitted** mode the operator turns **Block** on when **Reset** is off, the system may nondeterministically change **Overridden** to *true* or to *false*.

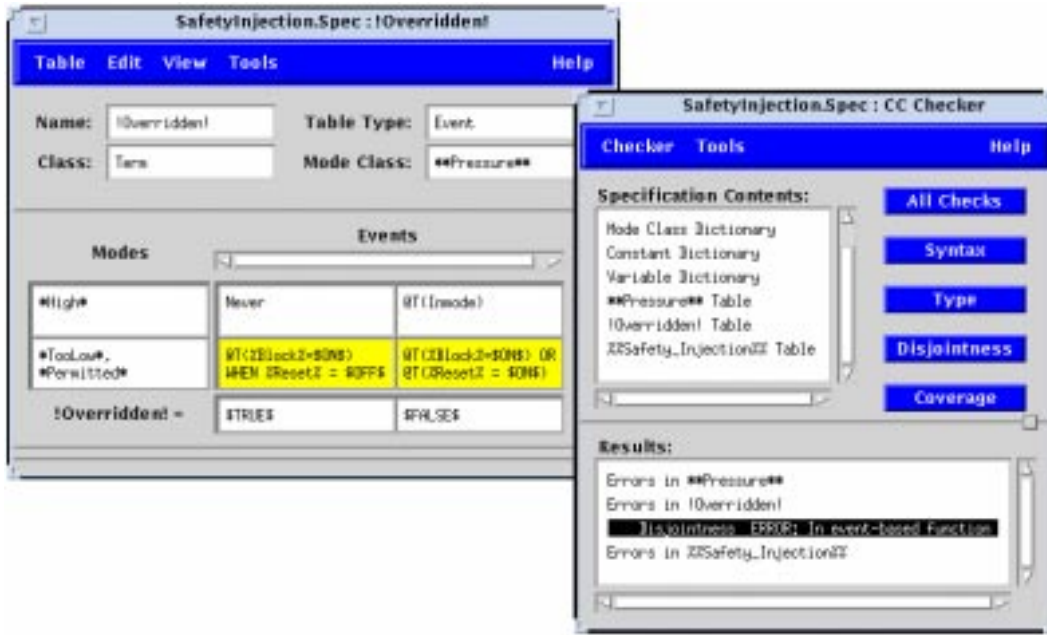


Figure 10: Disjointness Error in an Event Table.

## 6 Simulator

In a typical session with the simulator, the user begins with some “starting” state, that is, either the initial state or a state he defines.<sup>4</sup> To start a simulation, the user enters a sequence of one or more input events. To process each input event and thus simulate system execution, the user clicks on the Step Button (see the middle of Figure 11), and the simulator processes the next input event in the sequence. To compute each new state from an input event and the current state, the simulator applies  $T$ , the transform (i.e., next-state) function of our requirements model.

The simulator displays each system state in a window called the Simulator Display (see Figure 11). As each new state is computed, the simulator updates the Display window to reflect the new state. The simulator also supports a second window, called the Log window (see Figure 12). The Log shows the state history beginning with the “starting” state, organized into Monitored Variables and Dependent Variables. In the Log, the starting state is displayed in full; for each subsequent state, the Log lists the input event that caused the transition along with each mode class, term, and controlled variable whose value has changed.

In the example shown in Figure 11, the user has entered four events in the “Pending Events” area of the Display window. The upper portion of the Dis-

<sup>4</sup>Note that a starting state that differs from the initial state may not be reachable.

play window in Figure 11 shows the system state after the simulator has processed three of the four pending events (i.e., the user has clicked on the Step button three times). The Log in Figure 12 shows the system history after all four events have been processed.

To display the rule that caused a given entity to change value, the user double clicks on the entity and its value in the Log. The simulator then displays the table that defines the variable, highlighting the rule that led to the change. For example, double clicking on the expression “%%SafetyInjection%%=\$ON\$” near the bottom of the Log in Figure 12 will cause the simulator to display the table, shown in Figure 12 on the right, that caused **SafetyInjection** to change from **Off** to **On**. The simulator also highlights the rule that produced the change. In Figure 12, the highlighted rule is “If **Pressure** is **TooLow** and **Overridden** is *false*, then **SafetyInjection** is **On**.”

## 7 Verifier.

We are investigating the design of a state exploration tool that verifies application properties mechanically. Such a tool analyzes all states of a finite state machine model of a system to determine whether they satisfy selected properties. Our state-based requirements model and its transform function  $T$  provide a basis for building such a tool for verifying requirements specifications.

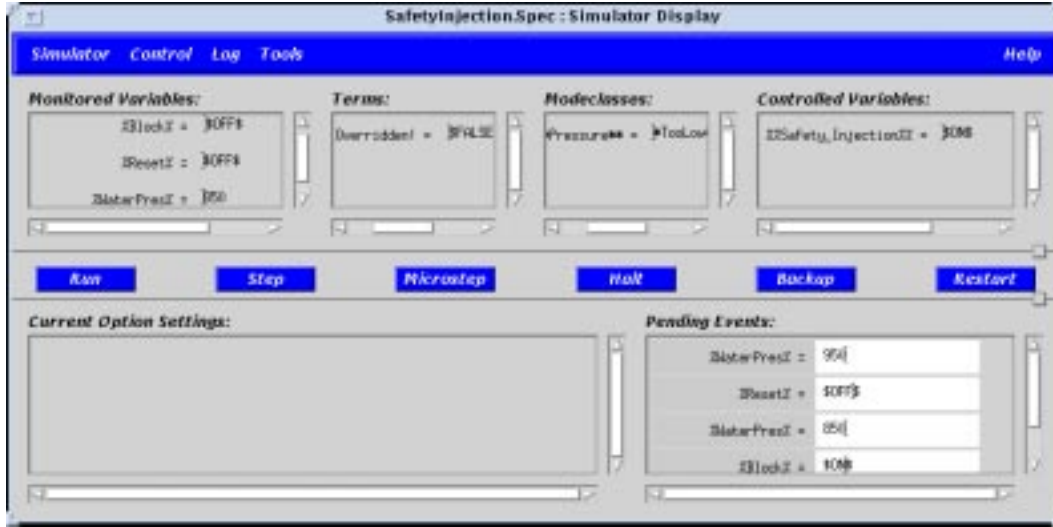


Figure 11: Simulator Display Showing Four Pending Events.



Figure 12: Simulator Log and Table with Rule Highlighted.

We are also investigating the linkage of our toolset with a mechanical proof system, such as EVES [17] or PVS [20], to support another form of verification—theorem proving. Such a tool can check formal proofs that the specifications satisfy properties of interest. Proofs would be done by hand using deductive reasoning (see, e.g., [11]) and checked mechanically using the proof system.

To illustrate application properties we wish to verify, some examples are listed. Each is a property of the simple control system described above.

1. If Block is Off and Pressure is TooLow, then SafetyInjection is On.
2. If Block is On and Reset is Off, then SafetyInjection is Off.

3. If WaterPres is greater than Low and WaterPres is less than Permit, then Pressure is Permitted.
4. If WaterPres' is greater than or equal to Low and Pressure is TooLow, then Pressure' is not equal to High.
5. If @T(WaterPres < Low) WHEN Block is Off, then SafetyInjection' is On.

## 8 Related Work.

In a related effort, Atlee and Gannon used model checking, a state exploration technique pioneered by Clarke [3], to test SCR requirements specifications for application properties [2]. Their tool analyzes proper-

ties defined in terms of mode classes and input variables only (e.g., Properties 3 and 4 above). Their state model is derived from the mode transition tables, extended by hand to incorporate the needed variable definitions. Such analysis is significant because it checks that the expressions in the mode transition tables capture desired properties. However, our requirements model provides the basis for a more general analysis: properties involving *all* of the entities can be checked against a state machine model of the complete specifications. That model is captured by our formal definitions of system and the transform function that maps an input event and the current state to the new state.

In other related work, Parnas describes ten small theorems related to his tabular notation (similar to other SCR notation) and challenges the developers of automated proof systems to prove the theorems [22]. Two of the theorems, the Domain Coverage Theorem and the Disjoint Domains Theorem, are slight variations of our Coverage and Disjointness properties. SRI researchers accepted Parnas' challenge. In a recent paper [23], they describe the mechanical proof of nine of Parnas' theorems using the "tcc-strategy" (tcc's are type-correctness conditions) of their proof system PVS [20]. Based on this result, we are investigating the utility of PVS and other mechanical provers for consistency checking.

## 9 Requirements Process

We envision the following process for developing requirements specifications for high assurance systems. Although such a process is an idealization of a real-world process, it shows how tools such as ours can be used incrementally to develop high assurance requirements specifications.

1. A formal notation, such as the SCR notation, is used to specify the requirements.
2. An automated consistency checker tests the specification for syntax and type correctness, coverage, determinism, and other application-independent properties.
3. The specification is executed symbolically using a simulator to ensure that it captures the customers' intent; the simulator can be run either manually as described above, or automatically using an input script (see, e.g., [4]).
4. In the later stages of requirements, mechanical support is used to analyze the specification for application properties. Initially, a small subset with fixed parameters and only a few states is extracted from the specification and a state exploration tool is used. This may be repeated, each time with a different or larger subset. Once there is sufficient confidence in

the specification, a deductive proof system may be used to help verify the complete specification or, more likely, safety-critical components.

## 10 Concluding Remarks

In our view, the use of properly designed software tools, in conjunction with a formal requirements notation, is an important step in developing high assurance systems. Such tools can

- **Liberate people to work on more creative tasks.** For example, although consistency checks are quite simple, the number of such checks needed in practical requirements specifications can be very large. In the Darlington plant certification, Parnas found that "reviewers spent too much of their time and energy checking for simple, application-independent properties" which distracted them from the "more difficult, safety-relevant issues" [22]. Automating such checks can save both the specifiers and reviewers considerable effort, thus liberating them to do more creative work.
- **Perform some tasks more effectively than people.** Tools can often find errors that people miss. In the experiments cited above, our tools found many significant errors overlooked by two independent review teams [13]. This does not indicate reviewer incompetence but illustrates instead that, for large specifications, tools are better than people for detecting certain classes of errors, such as missing cases and nondeterminism.
- **Increase confidence in the specification's correctness.** Analyzing the specifications with software tools can increase confidence that the specifications capture the required behavior. By symbolically executing the specifications using a simulator, the specifiers (and future users) can determine whether the external behavior represented by the specifications captures their intent. By running the verifier, the specifiers and the reviewers can check that the specification satisfies critical application properties.

We expect the process outlined above, which uses formal notation to specify requirements and computer-supported formal analysis to detect errors, to produce high quality requirements specifications. Such specifications are an important step toward developing high assurance systems.

## Acknowledgments

R. Jeffords is a major contributor to our requirements model. M. Archer, S. Faulk, R. Jeffords, and J. Kirby each provided many valuable suggestions for improving the paper. D. Kiskis and A. Rose contributed to the toolset's design and implementation.

## References

- [1] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software requirements for the A-7E aircraft. Technical Report NRL-9194, NRL, Wash., DC, 1992.
- [2] J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. In *Proc., ACM SIGSOFT Conf. on Software for Critical Systems*, New Orleans, December 1991.
- [3] E. M. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang. and Systems*, 8(2), April 1986.
- [4] P. Clements, C. Heitmeyer, B. Labaw, and A. Rose. MT: A toolset for specifying and analyzing real-time systems. In *Proc., Real-Time Systems Symp.*, Raleigh, NC, December 1993.
- [5] P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc., 15th Intern. Conf. on Software Eng.*, Baltimore, 1993.
- [6] D. Craigen et al. An international survey of industrial applications of formal methods. Technical Report NRL-9581, NRL, Wash., DC, 1993.
- [7] S. Faulk. *State Determination in Hard-Embedded Systems*. PhD thesis, Univ. of No. Carolina, Chapel Hill, 1989.
- [8] S. R. Faulk, J. Brackett, P. Ward, and J. Kirby. The CoRE method for real-time requirements. *IEEE Software*, 9(5), September 1992.
- [9] S. R. Faulk, L. Finneran, J. Kirby, S. Shah, and J. Sutton. Experience applying the CoRE method to the Lockheed C-130J. In *Proc., Ninth Annual Conf. on Computer Assurance*, Gaithersburg, MD, June 1994.
- [10] C. Heitmeyer, B. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proc., International Symposium on Requirements Engineering*, March 1995.
- [11] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A case study in formal verification of real-time systems. In *Proc., Real-Time Systems Symp.*, San Juan, Puerto Rico, December 1994.
- [12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical Report NRL-7499, NRL, Wash., DC, 1995. In preparation.
- [13] C. L. Heitmeyer and B. G. Labaw. Consistency checks for SCR-style requirements specifications. Technical Report 9586, NRL, Wash DC, December 1993.
- [14] C. L. Heitmeyer and J. McLean. Abstract requirements specifications: A new approach and its application. *IEEE Trans. Softw. Eng.*, SE-9(5), September 1983.
- [15] K. Heninger, D. Parnas, J. Shore, and J. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, NRL, Wash., DC, 1978.
- [16] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6(1), January 1980.
- [17] S. Kromodimoeljo, W. Pase, M. Saaltink, D. Craigen, and I. Meisels. A tutorial on EVES. Technical report, Odyssey Research Associates, Ottawa, Ont., 1993.
- [18] C. E. Landwehr, C. L. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Trans. on Comp. Syst.*, 2(3):198-222, August 1984.
- [19] Robin R. Lutz. Targeting safety-related errors during software requirements analysis. In *Proc., First ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Los Angeles, CA, December 1993.
- [20] S. Owre, N. Shankar, and J. Rushby. User guide for the PVS specification and verification system (Draft). Technical report, Computer Science Lab, SRI Intl., Menlo Park, CA, 1993.
- [21] D. Parnas and J. Madey. Functional documentation for computer systems engineering (Version 2). Technical Report CRL 237, Telecommunications Research Inst. of Ontario (TRIO), McMaster Univ., Hamilton, Ont., 1991.
- [22] D. L. Parnas. Some theorems we should prove. In *Proc., 1993 Intern. Conf. on HOL Theorem Proving and Its Applications*, Vancouver, BC, August 1993.
- [23] J. Rushby and M. Srivas. Using PVS to prove some theorems of David Parnas. In *Proc., 1993 Intern. Conf. on HOL Theorem Proving and Its Applications*, Vancouver, BC, August 1993.
- [24] A. J. van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application for monitoring systems. Technical Report TR 90-276, Queen's Univ., Kingston, Ont., 1990.
- [25] A. J. van Schouwen, D. L. Parnas, and J. Madey. Documentation of requirements for computer systems. In *Proc., RE'93 Requirements Symp.*, San Diego, January 1993.